Chapter 7: Object-Oriented Programming

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Code Shape

Please group program code as follows.

- Declare constants and variables
- Get input
- Calculate
- Display

Python is Object-Oriented (OOP)

In all the programs we have written until now, we have designed our program around functions: blocks of statements which manipulate data. This is called the procedure-oriented way of programming.

There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the object-oriented

programming paradigm. Most of the time you can use procedural or functional programming. When you are writing large programs or have a problem that is better suited to this method, you can use object-oriented programming techniques.

Python is an object-oriented programming language. Everything in Python is an object. We have been using many object-oriented concepts already. Object-oriented programming (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.

The key notion is that of an object. An object consists of two things: data (fields) and functions (called methods) that work with that data.

As an example, strings in Python are objects. The data of the string object is the actual characters that make up that string. The methods are things like lower, replace, and split. To Python, everything is an object. That includes not only strings and lists, but also integers, floats, and even functions themselves.

A Class Definition is a Blueprint for Objects

A class definition is a blueprint (set of plans) from which many individual objects can be constructed, created, produced, instantiated, or whatever verb you prefer to use for building something from a set of plans.

An object is a programming construct that encapsulates data and the ability to manipulate that data in a single software entity. The blueprint describes the data contained within and the behavior of objects instantiated according to the class definition.

An object's data is contained in variables defined within the class (often called member variables, instance variables, data members, attributes, fields, properties, etc.). The terminology used often depends on the background of the person writing the document.

Unlike Java and C++, once a Python object is instantiated, it can be modified in ways that no longer follow the blueprint of the class through the addition of new data members.

An object's behavior is controlled by methods defined within the class. In Python, methods are functions that are defined within a class definition.

An object is said to have state and behavior. At any instant in time, the state of an object is determined by the values stored in its variables and its behavior is determined by its methods.

What is an Object-Oriented Program?

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an Object?

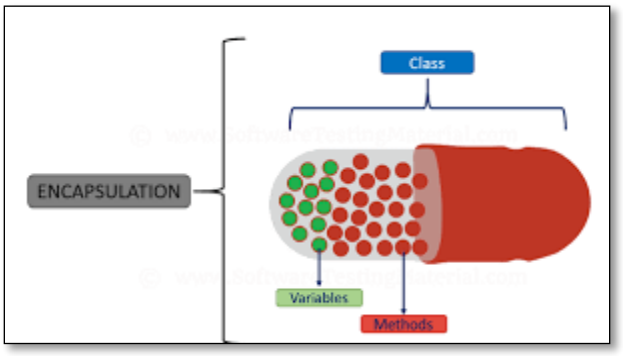
An object is a software construct that encapsulates data, along with the ability to use or modify that data.

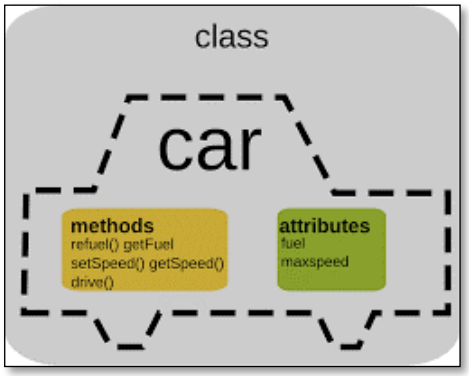
What is Encapsulation?

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

What is Data Hiding?

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.





Classes and Objects

One of the most important concepts in object-oriented programming is the distinction between classes and objects, which are defined as follows:

1. **Class** — A blueprint created by a programmer for an object. This defines a set of attributes that will characterize any object that is instantiated from this class.
2. **Object** — An instance of a class. A class is used to create an object in a program.

These are used to create patterns (in the case of classes) and then make use of the patterns (in the case of objects).

Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that **belong** to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the attributes of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called instance variables and class variables respectively.

A class is created using the **class** keyword. The fields and methods of the class are listed in an indented block.

The self Parameter

Class methods have only one specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list. This variable refers to the object itself, and by convention, it is given the name **self**.

The **self** parameter tells Python which object to operate on. **self** references the functions and class variables of that object.

In a sense, everything defined in an object belongs to it**self**.

Python Classes and Objects

At a basic level, an object is simply some code plus data structures that are smaller than a whole program. Defining a method allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain many methods as well as data that is manipulated by those methods. We call data items that are part of the object attributes.

We use the **class** keyword to define the data and code that will make up each of the objects. The **class** keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

Data Hiding: The _ (underscore) in front of the attribute name is to hide the data from anything outside the class or program. This would be considered private in other languages.

In Python, this is not enforced as in other languages, but is only a convention. Only the class and its methods are supposed to access and change the data directly.

The class is like a cookie cutter and the objects created using the class are the cookies. You don't put frosting on the cookie cutter; you put frosting on the cookies, and you can put different frosting on each cookie.

The following photo shows a class and two objects.

Create Class and Object

The simplest class possible is shown in the following example.

```
# Define a class
class Person:
    pass # An empty code block

# Create a Person() object named p
p = Person()

# Printing an object shows the memory address
print(p)
```

Example run:

```
<__main__.Person object at 0x0000028AC1B186A0>
```

How It Works

We create a new class using the **class** statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have

an empty block which is indicated using the **pass** statement. This is handy when sketching out the methods of a class.

We create an object/instance of this class using the name of the class followed by a pair of parentheses.

When you print an object directly, you get the memory address where your object is stored. The address will have a different value on your computer as Python will store the object wherever it finds space.

Methods

A method is a function that "belongs to" an object. **self** is the parameter that ties the method to an object. Method names usually include **verbs** since they represent **actions**.

We will now see an example.

```
class Person:
    # Method with the self parameter
    def say_hi(self):
        print("Hello, how are you? ")
Paul = Person()
Paul.say_hi()
```

Example run:

```
Hello, how are you?
```

How It Works

Here we see the self in action. Notice that the **say_hi()** method takes no parameters but still has the **self** in the function definition. **self** refers to the object. The object owns the method.

Getters and Setters

Getters and Setters allow the program to access the data attributes on the class. Getters and setters allow control over the values. You may validate the given value in the setter before setting the value.

- **getter:** returns the value of the attribute.
- **setter**: takes a parameter and assigns it to the attribute.

```
class Person:

    def get_hair_color(self):
        return self._hair_color

    def get_name(self):
        return self._name

    def set_hair_color(self, hair_color):
        self._hair_color = hair_color

    def set_name(self, name):
        self._name = name

    def say_hi(self):
        print(f"Hello, how are you? My name is {self._name}.")

Paul = Person()
Paul.set_hair_color("brown")
Paul.set_name("Paul")
Paul.say_hi()
print(f"My hair is {Paul.get_hair_color()}.")
```

Example run:

```
Hello, how are you? My name is Paul.
My hair is brown.
```

Tutorial 7.1 – Sammy the Shark

The following program demonstrates classes, objects, and methods.

An object is an instance of a class. We'll make a **Shark** object called **sammy**.

Create and name the program: **shark.py**

```
"""
    Name: shark_1.py
    Author:
    Created:
    Purpose: Demonstrate class, objects, methods,
    getters and setters
"""


class Shark:
    """ Define shark methods """

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def swim(self):
        print("The shark is swimming.")

    def be_awesome(self):
        print(f"{self._name} is being awesome.")


def main():
    # Create a shark object
    sammy = Shark()

    # Call shark methods
    sammy.set_name("Sammy")
    print(f"My name is {sammy.get_name()}")
    sammy.swim()
    sammy.be_awesome()


# Call the main function
if __name__ == "__main__":
    main()
```

How It Works

The **Shark** object **sammy** is using four methods:

- **get_name()** This returns the **self._name** attribute of the object.
- **set_name()** This sets the **self._name** attribute of the object from a passed in parameter.
- **swim()** and **be_awesome()** are methods which display text.

We call these methods using the dot operator (.), which is used to reference an attribute or method of the object. In this case, the attribute is a method and it's called with parentheses.

Because the keyword **self** was a parameter of the methods as defined in the **Shark** class, the **sammy** object gets passed to the methods. The **self** parameter ensures that the methods have a way of referring to object attributes.

When we call the methods, however, nothing is passed inside the parentheses, the object **sammy** is being automatically passed with the dot operator.

Example run:

```
My name is Sammy
The shark is swimming.
Sammy is being awesome.
```

The __init__ Method

The **__init__** method is used to define the initial state of an object.

This special method runs as soon as an object of a class is instantiated (i.e. created, constructed). This is called a constructor in other languages such as Java and C++.

This method is useful to do any initialization (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores at the beginning and at the end of the name. This indicates this is a dunder method. (double underscore)

```
# Filename: person_class.py
# Demonstration class

class Person:
    # init runs automatically when the object is created
    # This init method takes a single parameter: name
    def __init__(self, name):
        self._name = name
    # Define a method
    def say_hi(self):
        print(f"Hello, my name is {self._name}")

# Program starts here
barney = Person("Barney")
barney.say_hi()
```

Example run:

```
Hello, my name is Barney
```

How It Works

We define the **__init__** method as taking a parameter **name** (along with the usual self). We create a new field also called **self._name**.

When creating new instance **barney**, of the class **Person**, we do so by using the class name, followed by the arguments in the parentheses:

barney = Person("Barney")

We do not explicitly call the **__init__** method. It is called when the object is initialized.

Tutorial 7.2 – Constructing Sharks

The **__init__** method is used to initialize data. It is run as soon as an object of a class is instantiated.

Classes are useful because they allow us to create many similar objects based on the same blueprint. This program creates two objects from the same class.

Create the following program and name it: **shark_2.py**

```python
"""
    Name: shark_2.py
    Author:
    Created:
    Purpose: Demonstrate object construction
"""


class Shark:
    """
        Initialize the shark object with 2 parameters
        If parameters are not passed in, the default values are used
    """

    def __init__(self, name="Name", age=0):
        self._name = name
        self._age = age

    # Define shark methods
    def swim(self):
        print(f'{self._name} is swimming.')

    def be_awesome(self):
        print(f'{self._name} is being awesome.')

    def how_old(self):
        print(f'{self._name} is {self._age} years old.')

    # Getters and setters
    def get_name(self):
        return self._name

    def get_age(self):
        return self._age

    def set_name(self, name):
        self._name = name

    def set_age(self, age):
        self._age = age


def main():
    # Set name and age of Shark object during initialization
    sammy = Shark("Sammy", 2)
    sammy.how_old()
    sammy.be_awesome()

    # Create another shark object with default values
    stevie = Shark()
    stevie.set_name("Stevie")
    stevie.set_age(1)
    stevie.how_old()
    stevie.swim()


# Call the main method
if __name__ == "__main__":
    main()
```

Example run:

```
Sammy is 2 years old.
Sammy is being awesome.
Stevie is 1 years old.
Stevie is swimming.
```

How It Works

Notice the name being passed to the object. We defined the **__init__** method with the parameter name (along with the **self** keyword) and defined a variable within the method, name.

We also defined a new class variable, **_age**. To make use of age, we created a method in the class that calls for it. Constructor methods allow us to initialize certain attributes of an object.

The _ in front of the variable name **_age** hides the variable or makes it private to the class. This is called data hiding. We only allow access to object variables through the methods.

Because the constructor method is automatically initialized, we do not need to explicitly call it, only pass the arguments in the parentheses following the class name when we create a new instance of the class.

We created a second **Shark** object called **stevie** and passed the name "Stevie" to it. Classes make it possible to create more than one object following the same pattern without creating each one from scratch.

@property Decorator

We used this Person class previously to demonstrate getters and setters.

The Pythonic way to use getters and setters is with the **@property** decorator. This consolidates the getter and setter object calls. Instead of calling separate methods, we use the single property name to get or set the data attribute. Python determines what to do based on whether we are accessing or changing the property.

In the background, the data attributes are still considered private. Whenever we use **object.property**, Python internally calls the appropriate getter or setter.

```
class Person:
    def __init__(self, name, hair_color):
        self._name = name
        self._hair_color = hair_color

    @property
    def name(self):
        return self._name
    @name.setter
    def name(self, name):
        self._name = name

    @property
    def hair_color(self):
        return self._hair_color
    @hair_color.setter
    def hair_color(self, hair_color):
        self._hair_color = hair_color

    def say_hi(self):
        print(f"Hello, how are you? \nMy name is {self._name}.")

Paul = Person("Bill", "red")
# Assign value to property, setter
Paul.hair_color = "brown"
Paul.name = "Paul"
Paul.say_hi()
# Access value in property, getter
print(f"My hair is {Paul.hair_color}.")
```

Example run:

```
Hello, how are you?
My name is Paul.
My hair is brown.
```

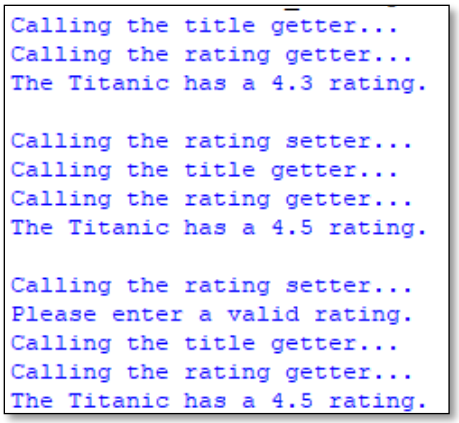
Tutorial 7.3 – Movie Class

Create a Python program named: **movie_class.py**

The movie class demonstrates @property getters and setters.

```python
"""
    Name: movie_class.py
    Author:
    Created:
    Purpose: Use @property Pythonic way of accessing attributes
"""


class Movie:

    def __init__(self, title, rating):
        self._title = title
        self._rating = rating

    # Define getter with @property decorator
    @property
    def rating(self):
        print("Calling the rating getter...")
        return self._rating

    # Define setter with @property decorator
    @rating.setter
    def rating(self, new_rating):
        print("Calling the rating setter...")
        # Data validation
        # rating must be between 1.0 and 5.0
        if 1.0 <= new_rating <= 5.0:
            self._rating = new_rating
        else:
            print("Please enter a valid rating.")


    # Define getter with @property decorator
    @property
    def title(self):
        print("Calling the title getter...")
        return self._title

    # Define setter with @property decorator
    @title.setter
    def title(self, new_title):
        print("Calling the title setter...")
        self._rating = new_title


favorite_movie = Movie("Titanic", 4.3)
print(f"The {favorite_movie.title} has a {favorite_movie.rating} rating.\n")

favorite_movie.rating = 4.5
print(f"The {favorite_movie.title} has a {favorite_movie.rating} rating.\n")

favorite_movie.rating = -5.6  # Invalid value
print(f"The {favorite_movie.title} has a {favorite_movie.rating} rating.")
```

Example run:



Classes in Separate Files

Classes and programs can be in separate files. A program can have several class files. This is very common in larger programs.

Tutorial 7.4 – Coin Flip

Coin Flip has two classes. The **Coin** class holds all the attributes and methods to flip a coin. The **coin_flip** program creates three coin objects and flips them.

Create a Python program named **coin.py**

```
"""
    Name: coin.py
    Author:
    Created:
    Purpose: The Coin class flips a coin
    and stores which side is up.
"""
# Import the random module
import random


class Coin:
    # The __init__ method initializes the
    # _sideup data attribute with "Heads"
    def __init__(self):
        self._sideup = "Heads"

    # The toss method generates a random number
    # in the range of 0 through 1. If the number
    # is 0, then sideup is set to "Heads".
    # Otherwise, sideup is set to "Tails".
    def toss(self):
        if random.randint(0, 1) == 0:
            self._sideup = "Heads"
        else:
            self._sideup = "Tails"

    # Returns the value referenced by sideup
    @property
    def sideup(self):
        return self._sideup
```

Create a Python program named **coin_flip.py** that uses the Coin class.

```
"""
    Name: coin_flip.py
    Author:
    Created:
    Purpose: Create 3 Coin objects
"""
# Import the coin class
import coin


def main():
    # Create three objects from the Coin class
    coin_1 = coin.Coin()
    coin_2 = coin.Coin()
    coin_3 = coin.Coin()

    # Display the side of each coin that is facing up
    print("\nI have three coins with these sides up:")
    print(coin_1.sideup)
    print(coin_2.sideup)
    print(coin_3.sideup)
    print()

    # Toss the coin
    print("I am flipping the coins...")
    print()
    coin_1.toss()
    coin_2.toss()
    coin_3.toss()

    # Display the side of each coin that is facing up
    print("Here is how the coins landed:")
    print(coin_1.sideup)
    print(coin_2.sideup)
    print(coin_3.sideup)
    print()


# Call the main function
if __name__ == "__main__":
    main()
```

Example run:

```
I have three coins with these sides up:
Heads
Heads
Heads

I am flipping the coins...

Here is how the coins landed:
Heads
Heads
Tails
```

repr() Function

The **repr()** function returns a printable representation of the given object.

```
class Person:
    name = "Adam"
    def __repr__(self):
        return repr(f"Hello {self._name}")

adam = Person()
print(repr(adam))
```

Default Arguments

When you define a function, you can specify a default value for each parameter.

```
def function_name(param1, param2=value2, param3=value3, ...):
```

When you call a function and pass an argument to the parameter that has a default value, the function will use that argument instead of the default value.

If you don't pass the argument, the function will use the default value.

```
class Person:
    def __init__(self, name="Adam"):
        self._name = name

    def __repr__(self):
        return repr(f"Hello {self._name}")

# Uses default argument
adam = Person()
print(repr(adam))
# Does not use default argument
fred = Person("Fred")
print(repr(fred))
```

Properties

Properties are a more Pythonic way of defining data attributes. You can have the same name for the setter and the getter. Python knows which on to use based on the direction of the data.

@property getter gets the value of the attribute.

@property.setter setter sets the value of the attribute.

The following example also demonstrates input validation with the setters.

Tutorial 7.5 - Property Decorators with Circles

```
"""
    Name: circle.py
    Author:
    Created:
    Purpose: Demonstrate the property decorator
"""


class Circle:
    # Class constant belongs to the class
    VALID_COLORS = ("Red", "Blue", "Green")

    def __init__(self, radius, color):
        print("Initializing a circle object")
        self._radius = radius
        self._color = color

    # Define getter with @property decorator
    @property
    def radius(self):
        print("Getting radius")
        return self._radius

    # Define setter with @property decorator
    @radius.setter
    def radius(self, new_radius):
        # Is new_radius an int and > 0
        if isinstance(new_radius, int) and new_radius > 0:
            print("Setting radius")
            self._radius = new_radius
        else:
            print("Please enter a valid radius.")

    # Define getter with @property decorator
    @property
    def color(self):
        print("Getting color")
        return self._color

    # Define setter with @property decorator
    @color.setter
    def color(self, new_color):
        # Is the color valid
        if new_color in Circle.VALID_COLORS:
            print("Setting color")
            self._color = new_color
        else:
            print("Please enter a valid color.")
```

```
51 my_circle = Circle(10, "Blue")
52
53 # Get radius
54 print(my_circle.radius)
55 # Set radius
56 my_circle.radius = 16
57 print(my_circle.radius)
58
59 my_circle.radius = 0
60 print(my_circle.radius)
61
62 # Get color
63 print(my_circle.color)
64
65 my_circle.color = "Red"
66 print(my_circle.color)
67
68 my_circle.color = "White"
69 print(my_circle.color)
```

Example run:

```
Initializing a circle object
Getting radius
10
Setting radius
Getting radius
16
Please enter a valid radius.
Getting radius
16
Getting color
Blue
Setting color
Getting color
Red
Please enter a valid color.
Getting color
Red
```

Pickle

Python provides a standard module called **pickle** which you can use to store any plain Python object in a file and then get it back later. This is called storing the object persistently. **pickling** is preserving data for later use, just like pickling beets preserves them for later yummy eating.

```
# Filename: shoplist_pickle.py

# Import the pickle module
import pickle

# The name of the file where we will store the serialized object
shoplist_file = "shoplist.pkl"

# List of things to buy
shoplist = ["apple", "mango", "carrot"]
print(shoplist)

# Write to the file
f = open(shoplist_file, "wb")

# Dump the object to a file
pickle.dump(shoplist, f)
print(f"shoplist written to {shoplist_file}")
f.close()

# Destroy the shoplist variable
del shoplist
print(f"shoplist destroyed in memory.")

# Read back from the storage
f = open(shoplist_file, "rb")
print(f"shoplist read back from {shoplist_file}")

# Load the object from the file
storedlist = pickle.load(f)
print(storedlist)

# Close the file
f.close()
```

Example run:

```
['apple', 'mango', 'carrot']
shoplist written to shoplist.pkl
shoplist destroyed in memory.
shoplist read back from shoplist.pkl
['apple', 'mango', 'carrot']
```

How It Works

To store an object in a file, **open** the file in write binary mode and then call the **dump** function of the **pickle** module. This process is called **pickling**.

We retrieve the object using the **load** function of the **pickle** module which returns the object. This process is called **unpickling**.

Inheritance

Another powerful feature of object-oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the parent class and the new class the child class.

For this example, we move our **PartyAnimal** class into its own file. Then, we can 'import' the **PartyAnimal** class in a new file and extend it, as follows:

```
# Import the PartyAnimal class from party.py
from party import PartyAnimal
# Defining and extending the
# CricketFan class based on PartyAnimal
# All PartyAnimal methods and data is available
class CricketFan(PartyAnimal):
   _points = 0
   # CricketFan method calling local and inherited party methods
   def six(self):
      self.__points = self.__points + 6
      # Inherited method from parent class
      self.party()
      print(self.__name,"points",self.__points)
sally = PartyAnimal("Sally")
sally.party()
jim = CricketFan("Jim")
jim.party()
jim.six()
# Display the attributes of the j object
# Some we defined, some are automatically defined when an object is created
print(dir(j))
```

When we define the **CricketFan** class, we indicate that we are extending the **PartyAnimal** class. This means that all the variables **(x)** and methods **(party)** from the **PartyAnimal** class are inherited by the **CricketFan** class. For example, within the **six** method in the **CricketFan** class, we call the party method from the **PartyAnimal** class.

As the program executes, we create **sally** and **jim** as independent instances of **PartyAnimal** and **CricketFan**. The **jim** object has additional capabilities beyond the **sally** object.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_s
ubclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas
shook__', '__weakref__', 'name', 'party', 'points', 'six', 'x']
```

In the **dir** output for the **jim** object (instance of the **CricketFan** class), we see that it has the attributes and methods of the parent class, as well as the attributes and methods that were added when the class was extended to create the **CricketFan** class.

Tutorial 7.6 - Parent and Child Fish (and Sharks)

Create the following Python program named: **fish.py**

```
'''
    Name: fish.py
    Author:
    Created:
    Purpose: Demonstrate Parent and Child classes
'''
class Fish:
    ''' Parent class '''
    # Initialize object and private class variables
    def __init__(self, first_name, last_name="Fish",
        # Hard code object variables
        skeleton="bone", eyelids=False):
        self.__first_name = first_name
        self.__last_name = last_name
        self.__skeleton = skeleton
        self.__eyelids = eyelids

    # Define class methods
    def swim(self):
        print(f'The fish is swimming.')
    def swim_backwards(self):
        print(f'The fish can swim backwards.')

    # Getters and setters
    def get_first_name(self):
        return self.__first_name
    def get_last_name(self):
        return self.__last_name
    def set_first_name(self, first_name):
        self.__first_name = first_name
    def set_last_name(self, last_name):
        self.__last_name = last_name

    def print_name(self):
        print(f'{self.__first_name} {self.__last_name}')

class Clownfish(Fish):
    ''' Child class '''
    def live_with_anemone(self):
        print(f'The clownfish is coexisting with sea anemone.')

class Shark(Fish):
    ''' Override the parent class '''
    def swim_backwards(self):
        print(f'The shark cannot swim backwards, but can sink backwards.')

```

```
47 def main():
48     # Create object with one argument
49     casey = Clownfish("Casey")
50     print(f'Create a clownfish named ')
51     print(f'{casey.get_first_name()} {casey.get_last_name()}')
52     casey.swim()
53     casey.live_with_anemone()
54     casey.swim_backwards()
55     # Create object with two arguments
56     shari = Shark("Shari", "the Shark")
57     print(f'Create a shark named Shari the Shark using two arguments')
58     shari.swim_backwards()
59     shari.set_last_name("the Salmon")
60     print(f'Change Shari\'s last name with a set method.')
61     shari.print_name()
62
63 # Call the main function
64 if __name__ == "__main__":
65     main()
```

Example run:

```
Create a clownfish named
Casey Fish
The fish is swimming.
The clownfish is coexisting with sea anemone.
The fish can swim backwards.
Create a shark named Shari the Shark using two arguments
The shark cannot swim backwards, but can sink backwards.
Change Shari's last name with a set method.
Shari the Salmon
```

Explanation

With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the pass keyword.

The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

The Clownfish class inherits all the attributes and methods from its parent, Fish. It has a special method will permit it to live with sea anemone.

The Shark method **swim_backwards()** prints a different string than the one in the Fish parent class because sharks are not able to swim backwards in the way that bony fish can.

Tutorial 7.7 - Convert Functional to OOP

In Chapter 5 we did a tutorial called **purchase_price_with_functions.py** This tutorial shows how to turn this into an OOP. Both programs give the same results. The OOP is much more expandable and simpler to use.

This is the original tutorial.

```
'''
    Name: purchase_price_with_functions.py
    Author:
    Created:
    Purpose: Calculate total sale with functions
'''

def main():
    '''    Return functions without arguments    '''
    purchase_price, quantity = get_input()

    # Return function with arguments
    total_sale = calculate_total_sale( purchase_price, quantity )

    # Void function
    display_sale( purchase_price, quantity, total_sale )

def get_input():
    '''    Get purchase price and quantity from the user    '''
    purchase_price = float(input('Enter the purchase price: '))
    quantity = int(input('Enter the quantity: '))
    # Return two values
    return purchase_price, quantity

def calculate_total_sale( price, quantity ):
    '''    Calculate the total sale    '''
    total_sale = price * quantity
    return total_sale

def display_sale( purchase_price, quantity, total_sale):
    '''
        Display information about the sale
        {"Purchase Price:":>15 formats string literal
        > align right 15 field width
    '''
    print (f'\n{"Purchase Price:":>15} ${purchase_price:,.2f}')
    print (f'{"Quantity:":>15} {quantity}')
    print (25 * '-')
    print (f'{"Total Sale:":>15} ${total_sale:,.2f}')

# If a standalone program, call the main function
# Else, use as a module
if __name__ == '__main__':
    main()
```

This is the same program written in OOP.

```
'''
    Name: purchase_price_oop.py
    Author:
    Created:
    Purpose: Calculate total sale with OOP
'''

class PurchasePrice():
    # Initializes object
    def __init__( self ):
        # Private class variables
        self.__purchase_price = 0
        self.__quantity = 0
        self.__total_sale = 0

    def get_input( self ):
        '''    Get purchase price and quantity from the user    '''
        self.__purchase_price = float(input('Enter the purchase price: '))
        self.__quantity = int(input('Enter the quantity: '))

    def calculate_total_sale( self ):
        '''    Calculate the total sale    '''
        self.__total_sale = self.__purchase_price * self.__quantity

    def display_sale( self ):
        '''
            Display information about the sale
            {"Purchase Price:":>15 formats string literal
            > align right 15 field width
        '''
        print (f'\n{"Purchase Price:":>15} ${self.__purchase_price:,.2f}')
        print (f'{"Quantity:":>15} {self.__quantity}')
        print (25 * '-')
        print (f'{"Total Sale:":>15} ${self.__total_sale:,.2f}')

def main():
    '''  Create a PurchasePrice() object   '''
    purchase_price = PurchasePrice()

    # Call object methods
    purchase_price.get_input()
    purchase_price.calculate_total_sale()
    purchase_price.display_sale()

# If a standalone program, call the main function
# Else, use as a module
if __name__ == '__main__':
    main()
```

Example run:

```
Enter the purchase price: 20
Enter the quantity: 2

Purchase Price: $20.00
      Quantity: 2
-------------------------
    Total Sale: $40.00
```

Designing an Object-Oriented Program

The first step is to identify the classes the program will need.

1. Identify the real-world objects, the nouns.

   Customer
   Address
   Car
   labor charges
   Toyota

   Some nouns contain other nouns. A Car can also be a Toyota. A customer would have an address. We need a Car and a Customer class.

2. Determine what describes or makes up the class, the data attributes.

   Customer: Name, Address, Phone

3. Determine the actions, the verbs, the methods.

   move_forward()
   make_purchase()
   create_invoice()

   Customer: set name, set address, get address

A class is responsible for one thing only. It should do that and do it well.

Glossary

attribute A variable that is part of a class.

class A template that can be used to construct an object. Defines the attributes and methods that will make up the object.

child class A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor An optional specially named method (__init__) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

destructor An optional specially named method (__del__) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional attributes and methods defined by the child class.

method A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object A constructed instance of a class. An object contains all the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangeably with 'object'.

parent class The class which is being extended to create a new child class. The parent class contributes all its methods and attributes to the new child class.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.